示例表

1 CREATE TABLE `employees` (

2  `id` int(11) NOT NULL AUTO\_INCREMENT,

3  `name` varchar(24) NOT NULL DEFAULT '' COMMENT '姓名',

4  `age` int(11) NOT NULL DEFAULT '0' COMMENT '年龄',

5  `position` varchar(20) NOT NULL DEFAULT '' COMMENT '职位',

6  `hire\_time` timestamp NOT NULL DEFAULT CURRENT\_TIMESTAMP COMMENT '入职时间',

7  PRIMARY KEY (`id`),

8  KEY `idx\_name\_age\_position` (`name`,`age`,`position`) USING BTREE

9 ) ENGINE=InnoDB AUTO\_INCREMENT=1 DEFAULT CHARSET=utf8 COMMENT='员工记录表';

10

11 INSERT INTO employees(name,age,position,hire\_time) VALUES('LiLei',22,'manager',NOW());

12 INSERT INTO employees(name,age,position,hire\_time) VALUES('HanMeimei', 23,'dev',NOW());

13 INSERT INTO employees(name,age,position,hire\_time) VALUES('Lucy',23,'dev',NOW());

**Mysql如何选择合适的索引**

mysql> EXPLAIN select \* from employees where name > 'a';



如果用name索引需要遍历name字段联合索引树，然后还需要根据遍历出来的主键值去主键索引树里再去查出最终数据，成本比全表扫描

还高，可以用覆盖索引优化，这样只需要遍历name字段的联合索引树就能拿到所有结果，如下：

mysql> EXPLAIN select name,age,position from employees where name > 'a' ;



mysql> EXPLAIN select \* from employees where name > 'zzz' ;



对于上面这两种 name>'a' 和 name>'zzz' 的执行结果，mysql最终是否选择走索引或者一张表涉及多个索引，mysql最

终如何选择索引，我们可以用trace工具来一查究竟，开启trace工具会影响mysql性能，所以只能临时分析sql使用，用

完之后立即关闭

trace工具用法：

1 mysql> set session optimizer\_trace="enabled=on",end\_markers\_in\_json=on; ‐‐开启trace

2 mysql> select \* from employees where name > 'a' order by position;

3 mysql> SELECT \* FROM information\_schema.OPTIMIZER\_TRACE;

4

5 查看trace字段：

6 {

7  "steps": [

8  {

9  "join\_preparation": { ‐‐第一阶段：SQL准备阶段

10  "select#": 1,

11  "steps": [

12  {

13  "expanded\_query": "/\* select#1 \*/ select `employees`.`id` AS `id`,`employees`.`name` AS `name`,`empl

oyees`.`age` AS `age`,`employees`.`position` AS `position`,`employees`.`hire\_time` AS `hire\_time` from

`employees` where (`employees`.`name` > 'a') order by `employees`.`position`"

14  }

15  ] /\* steps \*/

16  } /\* join\_preparation \*/

17  },

18  {

19  "join\_optimization": { ‐‐第二阶段：SQL优化阶段

20  "select#": 1,

21  "steps": [

22  {

23  "condition\_processing": { ‐‐条件处理

24  "condition": "WHERE",

25  "original\_condition": "(`employees`.`name` > 'a')",

26  "steps": [

27  {

28  "transformation": "equality\_propagation",

29  "resulting\_condition": "(`employees`.`name` > 'a')"

30  },

31  {

32  "transformation": "constant\_propagation",

33  "resulting\_condition": "(`employees`.`name` > 'a')"

34  },

35  {

36  "transformation": "trivial\_condition\_removal",

37  "resulting\_condition": "(`employees`.`name` > 'a')"

38  }

39  ] /\* steps \*/

40  } /\* condition\_processing \*/

41  },

42  {

43  "substitute\_generated\_columns": {

44  } /\* substitute\_generated\_columns \*/

45  },

46  {

47  "table\_dependencies": [ ‐‐表依赖详情

48  {

49  "table": "`employees`",

50  "row\_may\_be\_null": false,

51  "map\_bit": 0,

52  "depends\_on\_map\_bits": [

53  ] /\* depends\_on\_map\_bits \*/

54  }

55  ] /\* table\_dependencies \*/

56  },

57  {

58  "ref\_optimizer\_key\_uses": [

59  ] /\* ref\_optimizer\_key\_uses \*/

60  },

61  {

62  "rows\_estimation": [ ‐‐预估表的访问成本

63  {

64  "table": "`employees`",

65  "range\_analysis": {

66  "table\_scan": { ‐‐全表扫描情况

67  "rows": 10123, ‐‐扫描行数

68  "cost": 2054.7 ‐‐查询成本

69  } /\* table\_scan \*/,

70  "potential\_range\_indexes": [ ‐‐查询可能使用的索引

71  {

72  "index": "PRIMARY", ‐‐主键索引

73  "usable": false,

74  "cause": "not\_applicable"

75  },

76  {

77  "index": "idx\_name\_age\_position", ‐‐辅助索引

78  "usable": true,

79  "key\_parts": [

80  "name",

81  "age",

82  "position",

83  "id"

84  ] /\* key\_parts \*/

85  }

86  ] /\* potential\_range\_indexes \*/,

87  "setup\_range\_conditions": [

88  ] /\* setup\_range\_conditions \*/,

89  "group\_index\_range": {

90  "chosen": false,

91  "cause": "not\_group\_by\_or\_distinct"

92  } /\* group\_index\_range \*/,

93  "analyzing\_range\_alternatives": { ‐‐分析各个索引使用成本

94  "range\_scan\_alternatives": [

95  {

96  "index": "idx\_name\_age\_position",

97  "ranges": [

98  "a < name" ‐‐索引使用范围

99  ] /\* ranges \*/,

100  "index\_dives\_for\_eq\_ranges": true,

101  "rowid\_ordered": false, ‐‐使用该索引获取的记录是否按照主键排序

102  "using\_mrr": false,

103  "index\_only": false, ‐‐是否使用覆盖索引

104  "rows": 5061, ‐‐索引扫描行数

105  "cost": 6074.2, ‐‐索引使用成本

106  "chosen": false, ‐‐是否选择该索引

107  "cause": "cost"

108  }

109  ] /\* range\_scan\_alternatives \*/,

110  "analyzing\_roworder\_intersect": {

111  "usable": false,

112  "cause": "too\_few\_roworder\_scans"

113  } /\* analyzing\_roworder\_intersect \*/

114  } /\* analyzing\_range\_alternatives \*/

115  } /\* range\_analysis \*/

116  }

117  ] /\* rows\_estimation \*/

118  },

119  {

120  "considered\_execution\_plans": [

121  {

122  "plan\_prefix": [

123  ] /\* plan\_prefix \*/,

124  "table": "`employees`",

125  "best\_access\_path": { ‐‐最优访问路径

126  "considered\_access\_paths": [ ‐‐最终选择的访问路径

127  {

128  "rows\_to\_scan": 10123,

129  "access\_type": "scan", ‐‐访问类型：为scan，全表扫描

130  "resulting\_rows": 10123,

131  "cost": 2052.6,

132  "chosen": true, ‐‐确定选择

133  "use\_tmp\_table": true

134  }

135  ] /\* considered\_access\_paths \*/

136  } /\* best\_access\_path \*/,

137  "condition\_filtering\_pct": 100,

138  "rows\_for\_plan": 10123,

139  "cost\_for\_plan": 2052.6,

140  "sort\_cost": 10123,

141  "new\_cost\_for\_plan": 12176,

142  "chosen": true

143  }

144  ] /\* considered\_execution\_plans \*/

145  },

146  {

147  "attaching\_conditions\_to\_tables": {

148  "original\_condition": "(`employees`.`name` > 'a')",

149  "attached\_conditions\_computation": [

150  ] /\* attached\_conditions\_computation \*/,

151  "attached\_conditions\_summary": [

152  {

153  "table": "`employees`",

154  "attached": "(`employees`.`name` > 'a')"

155  }

156  ] /\* attached\_conditions\_summary \*/

157  } /\* attaching\_conditions\_to\_tables \*/

158  },

159  {

160  "clause\_processing": {

161  "clause": "ORDER BY",

162  "original\_clause": "`employees`.`position`",

163  "items": [

164  {

165  "item": "`employees`.`position`"

166  }

167  ] /\* items \*/,

168  "resulting\_clause\_is\_simple": true,

169  "resulting\_clause": "`employees`.`position`"

170  } /\* clause\_processing \*/

171  },

172  {

173  "reconsidering\_access\_paths\_for\_index\_ordering": {

174  "clause": "ORDER BY",

175  "steps": [

176  ] /\* steps \*/,

177  "index\_order\_summary": {

178  "table": "`employees`",

179  "index\_provides\_order": false,

180  "order\_direction": "undefined",

181  "index": "unknown",

182  "plan\_changed": false

183  } /\* index\_order\_summary \*/

184  } /\* reconsidering\_access\_paths\_for\_index\_ordering \*/

185  },

186  {

187  "refine\_plan": [

188  {

189  "table": "`employees`"

190  }

191  ] /\* refine\_plan \*/

192  }

193  ] /\* steps \*/

194  } /\* join\_optimization \*/

195  },

196  {

197  "join\_execution": { ‐‐第三阶段：SQL执行阶段

198  "select#": 1,

199  "steps": [

200  ] /\* steps \*/

201  } /\* join\_execution \*/

202  }

203  ] /\* steps \*/

204 }

205

206 结论：全表扫描的成本低于索引扫描，所以mysql最终选择全表扫描

207

208 mysql> select \* from employees where name > 'zzz' order by position;

209 mysql> SELECT \* FROM information\_schema.OPTIMIZER\_TRACE;

210

211 查看trace字段可知索引扫描的成本低于全表扫描，所以mysql最终选择索引扫描

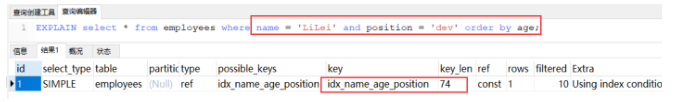
212

213 mysql> set session optimizer\_trace="enabled=off"; ‐‐关闭trace

**常见sql深入优化**

Order by与Group by优化

Case1：



分析：

利用最左前缀法则：中间字段不能断，因此查询用到了name索引，从key\_len=74也能看出，age索引列用

在排序过程中，因为Extra字段里没有using filesort

Case 2：

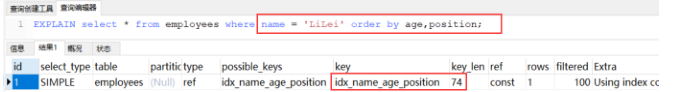


分析：

从explain的执行结果来看：key\_len=74，查询使用了name索引，由于用了position进行排序，跳过了

age，出现了Using filesort。

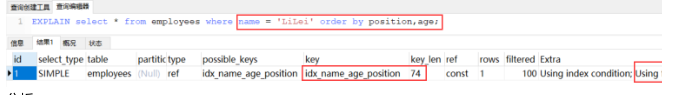
Case 3：



分析：

查找只用到索引name，age和position用于排序，无Using filesort。

Case 4：

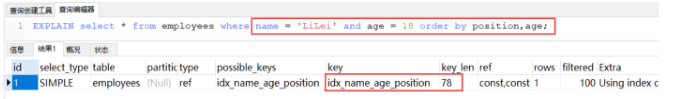


分析：

和Case 3中explain的执行结果一样，但是出现了Using filesort，因为索引的创建顺序为

name,age,position，但是排序的时候age和position颠倒位置了。

Case 5：



分析：

与Case 4对比，在Extra中并未出现Using filesort，因为age为常量，在排序中被优化，所以索引未颠倒，

不会出现Using filesort。

Case 6：

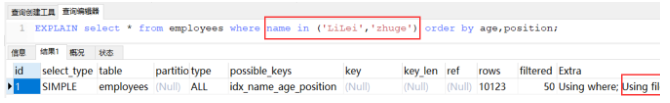


分析：

虽然排序的字段列与索引顺序一样，且order by默认升序，这里position desc变成了降序，导致与索引的

排序方式不同，从而产生Using filesort。Mysql8以上版本有降序索引可以支持该种查询方式。

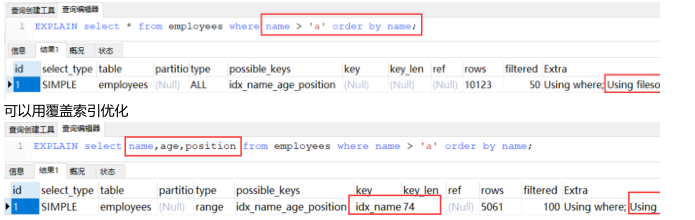
Case 7：



分析：

对于排序来说，多个相等条件也是范围查询

Case 8：

优化总结：

1、MySQL支持两种方式的排序filesort和index，Using index是指MySQL扫描索引本身完成排序。index

效率高，filesort效率低。

2、order by满足两种情况会使用Using index。

1) order by语句使用索引最左前列。

2) 使用where子句与order by子句条件列组合满足索引最左前列。

3、尽量在索引列上完成排序，遵循索引建立（索引创建的顺序）时的最左前缀法则。

4、如果order by的条件不在索引列上，就会产生Using filesort。

5、能用覆盖索引尽量用覆盖索引

6、group by与order by很类似，其实质是先排序后分组，遵照索引创建顺序的最左前缀法则。对于group by的优化如果不需要排序的可以加上order by null禁止排序。注意，where高于having，能写在where中的限定条件就不要去having限定了。

**Using filesort文件排序原理详解**

filesort文件排序方式

**单路排序**：是一次性取出满足条件行的所有字段，然后在sort buffer中进行排序；用trace工具可以看到sort\_mode信息里显示< sort\_key, additional\_fields >或者< sort\_key,packed\_additional\_fields >

**双路排序（又叫回表排序模式）**：是首先根据相应的条件取出相应的排序字段和可以直接定位行数据的行 ID，然后在 sort buffer 中进行排序，排序完后需要再次取回其它需要的字段；用trace工具可以看到sort\_mode信息里显示< sort\_key, rowid >

MySQL 通过比较系统变量 max\_length\_for\_sort\_data(默认1024字节) 的大小和需要查询的字段总大小来判断使用哪种排序模式。

如果 max\_length\_for\_sort\_data 比查询字段的总长度大，那么使用 单路排序模式；

如果 max\_length\_for\_sort\_data 比查询字段的总长度小，那么使用 双路排序模式。

示例验证下各种排序方式：

查看下这条sql对应trace结果如下(只展示排序部分)：

1 mysql> set session optimizer\_trace="enabled=on",end\_markers\_in\_json=on; ‐‐开启trace

2 mysql> select \* from employees where name = 'zhuge' order by position;

3 mysql> select \* from information\_schema.OPTIMIZER\_TRACE;

4

5 trace排序部分结果：

6 "join\_execution": { ‐‐Sql执行阶段

7  "select#": 1,

8  "steps": [

9  {

10  "filesort\_information": [

11  {

12  "direction": "asc",

13  "table": "`employees`",

14  "field": "position"

15  }

16  ] /\* filesort\_information \*/,

17  "filesort\_priority\_queue\_optimization": {

18  "usable": false,

19  "cause": "not applicable (no LIMIT)"

20  } /\* filesort\_priority\_queue\_optimization \*/,

21  "filesort\_execution": [

22  ] /\* filesort\_execution \*/,

23  "filesort\_summary": { ‐‐文件排序信息

24  "rows": 10000, ‐‐预计扫描行数

25  "examined\_rows": 10000, ‐‐参数排序的行

26  "number\_of\_tmp\_files": 3, ‐‐使用临时文件的个数，这个值如果为0代表全部使用的sort\_buffer内存排序，否则使用的磁盘文件排序

27  "sort\_buffer\_size": 262056, ‐‐排序缓存的大小

28  "sort\_mode": "<sort\_key, packed\_additional\_fields>" ‐‐排序方式，这里用的单路排序

29  } /\* filesort\_summary \*/

30  }

31  ] /\* steps \*/

32  } /\* join\_execution \*/

35 mysql> set max\_length\_for\_sort\_data = 10; ‐‐employees表所有字段长度总和肯定大于10字节

36 mysql> select \* from employees where name = 'zhuge' order by position;

37 mysql> select \* from information\_schema.OPTIMIZER\_TRACE;

38

39 trace排序部分结果：

40 "join\_execution": {

41  "select#": 1,

42  "steps": [

43  {

44  "filesort\_information": [

45  {

46  "direction": "asc",

47  "table": "`employees`",

48  "field": "position"

49  }

50  ] /\* filesort\_information \*/,

51  "filesort\_priority\_queue\_optimization": {

52  "usable": false,

53  "cause": "not applicable (no LIMIT)"

54  } /\* filesort\_priority\_queue\_optimization \*/,

55  "filesort\_execution": [

56  ] /\* filesort\_execution \*/,

57  "filesort\_summary": {

58  "rows": 10000,

59  "examined\_rows": 10000,

60  "number\_of\_tmp\_files": 2,

61  "sort\_buffer\_size": 262136,

62  "sort\_mode": "<sort\_key, rowid>" ‐‐排序方式，这里用的双路排序

63  } /\* filesort\_summary \*/

64  }

65  ] /\* steps \*/

66  } /\* join\_execution \*/

67

68

69 mysql> set session optimizer\_trace="enabled=off"; ‐‐关闭trace

我们先看单路排序的详细过程：

1. 从索引name找到第一个满足 name = ‘zhuge’ 条件的主键 id

2. 根据主键 id 取出整行，取出所有字段的值，存入 sort\_buffer 中

3. 从索引name找到下一个满足 name = ‘zhuge’ 条件的主键 id

4. 重复步骤 2、3 直到不满足 name = ‘zhuge’

5. 对 sort\_buffer 中的数据按照字段 position 进行排序

6. 返回结果给客户端

我们再看下双路排序的详细过程：

1. 从索引 name 找到第一个满足 name = ‘zhuge’  的主键id

2. 根据主键 id 取出整行，把排序字段 position 和主键 id 这两个字段放到 sort buffer 中

3. 从索引 name 取下一个满足 name = ‘zhuge’  记录的主键 id

4. 重复 3、4 直到不满足 name = ‘zhuge’

5. 对 sort\_buffer 中的字段 position 和主键 id 按照字段 position 进行排序

6. 遍历排序好的 id 和字段 position，按照 id 的值回到原表中取出 所有字段的值返回给客户端

其实对比两个排序模式，单路排序会把所有需要查询的字段都放到 sort buffer 中，而双路排序只会把主键

和需要排序的字段放到 sort buffer 中进行排序，然后再通过主键回到原表查询需要的字段。

如果 MySQL 排序内存配置的比较小并且没有条件继续增加了，可以适当把 max\_length\_for\_sort\_data 配

置小点，让优化器选择使用双路排序算法，可以在sort\_buffer 中一次排序更多的行，只是需要再根据主键

回到原表取数据。

如果 MySQL 排序内存有条件可以配置比较大，可以适当增大 max\_length\_for\_sort\_data 的值，让优化器

优先选择全字段排序(单路排序)，把需要的字段放到 sort\_buffer 中，这样排序后就会直接从内存里返回查

询结果了。

所以，MySQL通过 max\_length\_for\_sort\_data 这个参数来控制排序，在不同场景使用不同的排序模式，

从而提升排序效率。

注意，如果全部使用sort\_buffer内存排序一般情况下效率会高于磁盘文件排序，但不能因为这个就随便增

大sort\_buffer(默认1M)，mysql很多参数设置都是做过优化的，不要轻易调整。

**分页查询优化**

1 示例表：

2 CREATE TABLE `employees` (

3  `id` int(11) NOT NULL AUTO\_INCREMENT,

4  `name` varchar(24) NOT NULL DEFAULT '' COMMENT '姓名',

5  `age` int(11) NOT NULL DEFAULT '0' COMMENT '年龄',

6  `position` varchar(20) NOT NULL DEFAULT '' COMMENT '职位',

7  `hire\_time` timestamp NOT NULL DEFAULT CURRENT\_TIMESTAMP COMMENT '入职时间',

8  PRIMARY KEY (`id`),

9  KEY `idx\_name\_age\_position` (`name`,`age`,`position`) USING BTREE

10 ) ENGINE=InnoDB AUTO\_INCREMENT=1 DEFAULT CHARSET=utf8 COMMENT='员工记录表';

11

12 drop procedure if exists insert\_emp;

13 delimiter ;;

14 create procedure insert\_emp()

15 begin

16   declare i int;

17   set i=1;

18   while(i<=100000)do

19     insert into employees(name,age,position) values(CONCAT('zhuge',i),i,'dev');

20     set i=i+1;

21   end while;

22 end;;

23 delimiter ;

24 call insert\_emp();

**很多时候我们业务系统实现分页功能可能会用如下sql实现**

1 mysql> select \* from employees limit 10000,10;

表示从表  employees  中取出从 10001 行开始的 10 行记录。看似只查询了 10 条记录，实际这条 SQL 是先读取 10010条记录，然后抛弃前 10000 条记录，然后读到后面 10 条想要的数据。因此要查询一张大表比较靠后的数据，执行效率

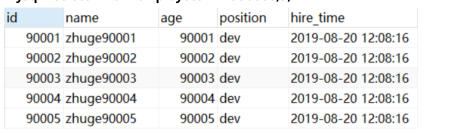
是非常低的。

>>常见的分页场景优化技巧：

1、根据自增且连续的主键排序的分页查询

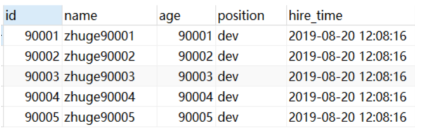
首先来看一个根据自增且连续主键排序的分页查询的例子：

mysql> select \* from employees limit 90000,5;



该 SQL 表示查询从第 90001开始的五行数据，没添加单独 order by，表示通过主键排序。我们再看表  employees  ，因为主键是自增并且连续的，所以可以改写成按照主键去查询从第 90001开始的五行数据，如下：

mysql> select \* from employees where id > 90000 limit 5;



查询的结果是一致的。我们再对比一下执行计划：

mysql> EXPLAIN select \* from employees limit 90000,5;

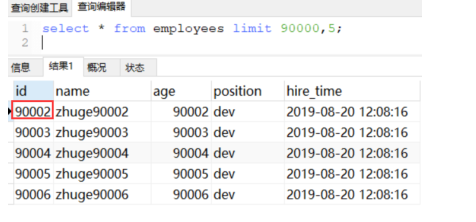




显然改写后的 SQL 走了索引，而且扫描的行数大大减少，执行效率更高。

但是，这条 改写的SQL 在很多场景并不实用，因为表中可能某些记录被删后，主键空缺，导致结果不一致，如下图试验

所示（先删除一条前面的记录，然后再测试原 SQL 和优化后的 SQL）：





两条 SQL 的结果并不一样，因此，如果主键不连续，不能使用上面描述的优化方法。

另外如果原 SQL 是 order by 非主键的字段，按照上面说的方法改写会导致两条 SQL 的结果不一致。所以这种改写得满

足以下两个条件：

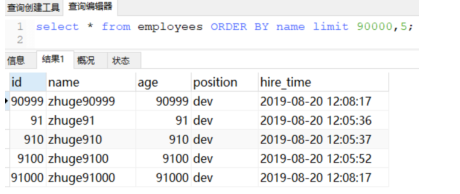
主键自增且连续

结果是按照主键排序的

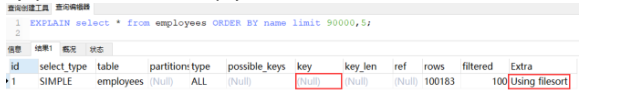
2、根据非主键字段排序的分页查询

再看一个根据非主键字段排序的分页查询，SQL 如下：

mysql>  select \* from employees ORDER BY name limit 90000,5;



mysql> EXPLAIN select \* from employees ORDER BY name limit 90000,5;



发现并没有使用 name 字段的索引（key 字段对应的值为 null），具体原因上节课讲过：扫描整个索引并查找到没索引

的行(可能要遍历多个索引树)的成本比扫描全表的成本更高，所以优化器放弃使用索引。

知道不走索引的原因，那么怎么优化呢？

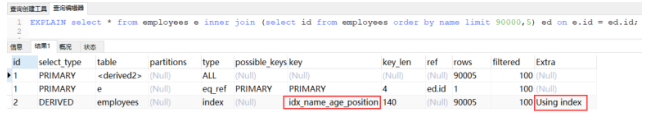
其实关键是让排序时返回的字段尽可能少，所以可以让排序和分页操作先查出主键，然后根据主键查到对应的记录，SQL

改写如下

mysql> select \* from employees e inner join (select id from employees order by name limit 90000,5) ed on e.id = ed.id;



需要的结果与原 SQL 一致，执行时间减少了一半以上，我们再对比优化前后sql的执行计划：



原 SQL 使用的是 filesort 排序，而优化后的 SQL 使用的是索引排序。

**Join关联查询优化**

1 示例表：

2 CREATE TABLE `t1` (

3  `id` int(11) NOT NULL AUTO\_INCREMENT,

4  `a` int(11) DEFAULT NULL,

5  `b` int(11) DEFAULT NULL,

6  PRIMARY KEY (`id`),

7  KEY `idx\_a` (`a`)

8 ) ENGINE=InnoDB AUTO\_INCREMENT=10001 DEFAULT CHARSET=utf8;

9

10 create table t2 like t1;

11

12 往t1表插入1万行记录，往t2表插入100行记录

**mysql的表关联常见有两种算法**

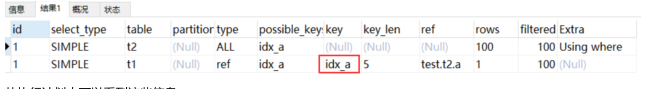
Nested-Loop Join 算法

Block Nested-Loop Join 算法

1**、 嵌套循环连接 Nested-Loop Join(NLJ) 算法**

一次一行循环地从第一张表（称为驱动表）中读取行，在这行数据中取到关联字段，根据关联字段在另一张表（被驱动表）里取出满足条件的行，然后取出两张表的结果合集。

mysql> EXPLAIN select\*from t1 inner join t2 on t1.a= t2.a;



从执行计划中可以看到这些信息：

驱动表是 t2，被驱动表是 t1。先执行的就是驱动表(执行计划结果的id如果一样则按从上到下顺序执行sql)；优化器一般会优先选择小表做驱动表。所以使用 inner join 时，排在前面的表并不一定就是驱动表。

使用了 NLJ算法。一般 join 语句中，如果执行计划 Extra 中未出现 Using join buffer 则表示使用的 join 算法是 NLJ。

上面sql的大致流程如下：

1. 从表 t2 中读取一行数据；

2. 从第 1 步的数据中，取出关联字段 a，到表 t1 中查找；

3. 取出表 t1 中满足条件的行，跟 t2 中获取到的结果合并，作为结果返回给客户端；

4. 重复上面 3 步。

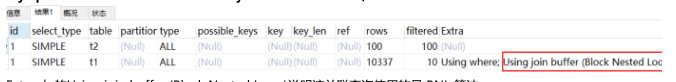
整个过程会读取 t2 表的所有数据(扫描100行)，然后遍历这每行数据中字段 a 的值，根据 t2 表中 a 的值索引扫描 t1 表中的对应行(扫描100次 t1 表的索引，1次扫描可以认为最终只扫描 t1 表一行完整数据，也就是总共 t1 表也扫描了100行)。因此整个过程扫描了 200 行。

如果被驱动表的关联字段没索引，使用NLJ算法性能会比较低(下面有详细解释)，mysql会选择Block Nested-Loop Join算法。

**2、 基于块的嵌套循环连接 Block Nested-Loop Join( BNL )算法**

把驱动表的数据读入到 join\_buffer 中，然后扫描被驱动表，把被驱动表每一行取出来跟 join\_buffer 中的数据做对比。

mysql>EXPLAIN select\*from t1 inner join t2 on t1.b= t2.b;



Extra 中 的Using join buffer (Block Nested Loop)说明该关联查询使用的是 BNL 算法。

上面sql的大致流程如下：

1. 把 t2 的所有数据放入到 join\_buffer 中

2. 把表 t1 中每一行取出来，跟 join\_buffer 中的数据做对比

3. 返回满足 join 条件的数据

整个过程对表 t1 和 t2 都做了一次全表扫描，因此扫描的总行数为10000(表 t1 的数据总量) + 100(表 t2 的数据总量) =10100。并且 join\_buffer 里的数据是无序的，因此对表 t1 中的每一行，都要做 100 次判断，所以内存中的判断次数是100 \* 10000= 100 万次。被驱动表的关联字段没索引为什么要选择使用 BNL 算法而不使用 Nested-Loop Join 呢？

如果上面第二条sql使用 Nested-Loop Join，那么扫描行数为 100 \* 10000 = 100万次，这个是磁盘扫描。

很显然，用BNL磁盘扫描次数少很多，相比于磁盘扫描，BNL的内存计算会快得多。

因此MySQL对于被驱动表的关联字段没索引的关联查询，一般都会使用 BNL 算法。如果有索引一般选择 NLJ 算法，有索引的情况下 NLJ 算法比 BNL算法性能更高

对于关联sql的优化关联字段加索引，让mysql做join操作时尽量选择NLJ算法

小标驱动大表，写多表连接sql时如果明确知道哪张表是小表可以用**straight\_join**写法固定连接驱动方式，省去mysql优化器自己判断的时间

straight\_join解释：straight\_join功能同join类似，但能让左边的表来驱动右边的表，能改表优化器对于联表查询的执行顺序。

比如：select \* from t2 straight\_join t1 on t2.a = t1.a; 代表制定mysql选着 t2 表作为驱动表。

straight\_join只适用于inner join，并不适用于left join，right join。（因为left join，right join已经代表指定了表的执行顺序）

尽可能让优化器去判断，因为大部分情况下mysql优化器是比人要聪明的。使用straight\_join一定要慎重，因为部分情况下人为指定的执行顺序并不一定会比优化引擎要靠谱。

**in和exsits优化**

原则：小表驱动大表，即小的数据集驱动大的数据集

**in**：时会先执行in（）括号内的子查询，然后根据结果集查询主select

**exists**：时会先执行主查询，将主查询的结果集与exists（）括号内的子查询进行比对，返回最终的结果集

当用于判断a表中的数据在b表中是否存在时，试用表连接效率较低

in：当B表的数据集小于A表的数据集时，in优于exists

1 select \* from A where id in (select id from B)

2 #等价于：

3   for(select id from B){

4  select \* from A where A.id = B.id

5  }

exists：当A表的数据集小于B表的数据集时，exists优于in

　　将主查询A的数据，放到子查询B中做条件验证，根据验证结果（true或false）来决定主查询的数据是否保留

1 select \* from A where exists (select 1 from B where B.id = A.id)

2 #等价于:

3  for(select \* from A){

4  select \* from B where B.id = A.id

5  }

6

7 #A表与B表的ID字段应建立索引

1、EXISTS (subquery)只返回TRUE或FALSE,因此子查询中的SELECT \* 也可以用SELECT 1替换,官方说法是实际执行时会

忽略SELECT清单,因此没有区别

2、EXISTS子查询的实际执行过程可能经过了优化而不是我们理解上的逐条对比

3、EXISTS子查询往往也可以用JOIN来代替（一般效率较低），何种最优需要具体问题具体分析

**count(\*)查询优化**

# 临时关闭mysql查询缓存，为了查看sql多次执行的真实时间

mysql> set global query\_cache\_size=0;

mysql> set global query\_cache\_type=0;

mysql> EXPLAIN select count(1) from employees;

mysql> EXPLAIN select count(id) from employees;

mysql> EXPLAIN select count(name) from employees;

mysql> EXPLAIN select count(\*) from employees;



**Select count(id):扫描整个主键索引树，数据较二级索引多，把所有数据取出并进行计算数据行数**

**Select count(name):name为辅助索引字段，则扫描索引拿出name字段的值并计算行数，如果有null值不进行加1运算**

**Select count（\*）:使用辅助索引树，性能与select count（name）性能接近**

**Select count (1)：使用表的辅助索引，扫描辅助索引树但不取出索引字段的值，仅进行加1操作**

四种操作的性能比较：1>name>\*>id

Mysql5.7之后的版本对其进行了优化，主键索引被优化为使用辅助索引，四种方式接近，建议使用count(\*)

四个sql的执行计划一样，说明这四个sql执行效率应该差不多，区别在于根据某个字段count不会统计字段为null值的数据行

为什么mysql最终选择辅助索引而不是主键聚集索引？因为二级索引相对主键索引存储数据更少，检索性能应该更高

常见优化方法

1、查询mysql自己维护的总行数

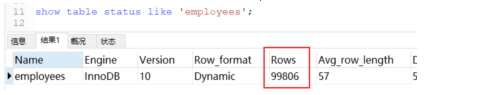
对于myisam存储引擎的表做不带where条件的count查询性能是很高的，因为myisam存储引擎的表的总行数会被mysql存储在磁盘上，查询不需要计算



对于innodb存储引擎的表mysql不会存储表的总记录行数，查询count需要实时计算

2、show table status

如果只需要知道表总行数的估计值可以用如下sql查询，性能很高



3、将总数维护到Redis里

插入或删除表数据行的时候同时维护redis里的表总行数key的计数值(用incr或decr命令)，但是这种方式可能不准，很难

保证表操作和redis操作的事务一致性

4、增加计数表

插入或删除表数据行的时候同时维护计数表，让他们在同一个事务里操作